The Terminator to Android Hardening Services

Yueqian Zhang, Xiapu Luo, Haoyang Yin
Department of Computing
The Hong Kong Polytechnic University
Percentage of top 10 apps in each category which have repacked version:

- 100% of the apps of Widgets, Media & Video, etc.
- 90% of the apps of Business, Music & Audio, etc.

Source: Trend Micro
Android 及 ARM 原生語言 [第二版]
逆向工程破解 Android APP 安全
別讓你的程式碼成為別人的砲灰
Outline

- Background
- DexHunter
- Analysis of major products
- Related work
Outline

- Background
- DexHunter
- Analysis of major products
- Related work
Dex File

- Java source code -> Java class
  - Java class: each file contains one class
  - dex: one file contains all classes

- Reorganize constant pools in each class file into shared and type-specific constant pools
Dex File

❖ The executable of an App.
❖ The header contains the length and the offset for each section.
❖ `class_defs` section contains `class_def_items`, each of which describes a class.
A `class_def_item` points to a `class_data_item`.

A `class_data_item` contains the data of a class.

Each method is described by an `encoded_method`.

An `encoded_method` points to a `code_item`.

A `code_item` contains the instructions of a method.
OAT File

- It is generated while an app is installed or a jar file is loaded.

- `/frameworks/base/services/java/com/android/server/pm/PackageManagerService.java`
- Constructor method → `scanDirLI()` → `scanPackageLI()` → `performDexOptLI()` → `mInstaller.dexopt()`

- It is an ELF file.

```
system@priv-app@VoiceDialer.apk@classes.dex: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (GNU/Linux), dynamically linked, stripped
```
OAT File

- Three symbols in dynamic section.
  - oatdata
  - oatexec
  - oatlastword

- The original dex file is contained in the **oatdata** section.

- The compiled native instructions are contained in the **oatexec** section.
Outline

- Background
- **DexHunter**
  - Where to unpack the app?
  - When to unpack the app?
  - How to unpack the app?
- Analysis of major products
- Related work
Where to dump dex file?

- Four occasions
  - Opening a Dex file;
  - Loading a class;
  - Initializing a class;
  - Invoking a method;
Opening a Dex File

Operations

- Open an APK file;
- Check whether it has been cached;
- If not, extract the dex file from the APK and generate the cached dex file;
- Open the cached dex file.
Procedure of Opening a Dex File in ART

1. Class loader load and open a dex file
2. Find it from the cache map and return
   - Yes
   - No
3. Has it been opened?
   - Yes
   - No
   - Find its corresponding oat file in the output path
     - Yes
     - No
     - Generating an oat file from the input dex file
4. Does it exist?
   - Yes
   - No
   - Generating an oat file from the input dex file
5. Does it succeed?
   - Yes
   - No
   - Fail and throw an exception
Loading a Class

Operations

- Form a class object from the data;
- Verify the legitimacy of access flags and the data;
- Populate all fields in the class object;
- Deal with its super classes and/or interfaces;
- Conduct some other checking.
Two Ways of Loading a Classes

- **Explicit approach**
  - `Class.forName()`, `ClassLoader.loadClass()`.

- **Implicit approach**
  - E.g., `new` operation, accessing static members, etc.
Implementation in ART

- **Explicit**
  - `ClassLoader.loadClass` → `DexFile_defineClassNative`
  - `Class.forName` → `Class_classForName`

- **Implicit**
  - `{new operations and so on}` → `artAllocObjectFromCode`
Implementation in ART
Implementation in DVM

Explicit

- `ClassLoader.loadClass` → `Dalvik_dalvik_system_DexFile`
- `defineClassNative`
- `Class.forName` → `Dalvik_java_lang_Class_classForName`

Implicit

- `new` operations and so on → `dvmResolveClass`
Implementation in DVM
Class Loaders at Java Level

Three class loaders

- **BootClassLoader**
  - It is used for loading system classes.

- **DexClassLoader**
  - It is used for loading external files.

- **PathClassLoader**
  - It is used by the framework.
Inheritance Relationship
Parent Delegation Model

```java
Class<?> loadClass(String className, boolean resolve {
    Class<?> clazz = findLoadedClass(className);
    if (clazz == null) {
        clazz = parent.loadClass(className, false);
    }
    if (clazz == null) {
        clazz = findClass(className);
    }
    return clazz;
}
```
Each subclass of ClassLoader implements its own `findClass()`. Each subclass of ClassLoader inherits `loadClass()` except `BootClassLoader`.
Differences between Java and Android

- `defineClass()` in `ClassLoader` (Android) is not implemented.
  - Throw `UnsupportedOperationException`

- `URLClassLoader` in Android also cannot load a class, because
  - `URLClassLoader.findClass() → URLHandler/URLJarHandler.findClass() → createClass () → ClassLoader.defineclass()`
A Loaded Class Object in ART
A Loaded Class Object in DVM
When does Initializing Classes happen?

- Before the class object is used;
- Before the first static data member is accessed;
- Before the first static method is invoked;
- Before the first instance is generated;
- ...

Invoking a Method

- DVM or ART interpreting mode
  - Execute the instructions in the `code_item`.

- ART native mode
  - Execute the native instructions in `oatexec` section.
When to unpack the app?

When the first class of the app is being loaded.

Why?

Before a class is loaded, the content of the class should be available in the memory;

When the class is initialized, some content in memory may be modified dynamically;

Just before a method is invoked, its code_item or instructions should be available.

How?

Load and initialize all classes proactively.
How to unpack the apk?

- Integrate our tool into Android runtime including DVM and ART.
- Wait for the proper occasion.
- Locate the target memory region.
- Dump the selected memory.
- Correct and reconstruct the dex file.
DexHunter

Memory Space → Target Region

- Locate

Target Region:
- Split
- Parsing each class
- Collect
- Copy each class_def_item and correct if needed
- Write

DexClassData:
- Collected code_item
- Class_def_item
- Write

Collected DexClassData:
- Extra

Part1:
- Data
Loading & Initializing Classes

- Traverse all `class_def_items` in the dex file.
- For each one, we load it with `FindClass` function (ART) or `dvmDefineClass` function (DVM).
- Then we initialize it with `EnsureInitialized` function (ART) or `dvmIsClassInitialized` & `dvmInitClass` functions (DVM).
Locating the Target Memory Region

- The target memory region contains the dex file.

- We use a special string to determine whether the current dex file is what we want.
The Special String in ART

◊ ART: the string “location_” in DexFile objects.

◊ The opened apk file’s path →
  *dex_file_location* in generated oat file’s header
  → *dex_file_location_* in OatDexFile objects
  → *location_* in DexFile objects by function *DexFile::Open*
The Special String in DVM

확정 DVM: the string “fileName” in DexOrJar objects.

확정 The opened apk file path \(\rightarrow\)
fileName in DexOrJar objects by function
\textit{Dalvik}\_dalvik\_system\_DexFile\_openDexFileNative.

확정 For Dalvik\_dalvik\_system\_DexFile\_openDexFile\_bytearray, fileName is always equal to “\textit{<memory>}”.
Extracting the Dex File in Memory

- Divide the target memory region
  - Part 1: the content before the `class_defs` section
  - Part 2: the `class_defs` section
  - Part 3: the content after the `class_defs` section

- Dump part 1 into a file named `part1` and part 3 into a file named `data`. 
Parsing the Content

❖ Parse `class_defs` section.
❖ Getting each `class_data_item` from `class_def_item`.
❖ Read the corresponding content into a `DexClassData` object.
❖ Notice: some fields in a `class_data_item` are encoded by LEB128 algorithm.

```c
struct DexClassData { // For one class_def_item
    DexClassDataHeader header;
    DexField* staticFields;
    DexField* instanceFields;
    DexMethod* directMethods;
    DexMethod* virtualMethods;
};

struct DexField { // For one field
    uint32_t delta_fieldIdx;
    uint32_t accessFlags;
};

struct DexClassDataHeader { // For one header
    uint32_t staticFieldsSize;
    uint32_t instanceFieldsSize;
    uint32_t directMethodsSize;
    uint32_t virtualMethodsSize;
};

struct DexMethod { // For one method
    uint32_t delta_methodIdx;
    uint32_t accessFlags;
    uint32_t codeOff;
};
```
Correcting and Collecting

Why?

◧ Packing services may modify the memory dynamically.

◧ The memory consists of the region containing the dex file and the method objects (i.e., \textit{ArtMethod} in ART, \textit{Method} in DVM) managed by runtime.

◧ The runtime executes instructions according to the managed method objects.
Correcting and Collecting

- We check each:
  - `class_data_off` in `class_def_item`.
  - `accessflag` and `codeoff` in `DexMethod` of parsed `class_data_item` (i.e., `DexClassData` object).
How?

- Determine whether the `class_data_off` in `class_def_item` exists in the scope of the dex file.
  - Copy all `class_def_items` and write them into a file named `classdef`.
  - Collect the outside `class_data_items` into a file named `extra`.

- Correct the fields in selected `DexClassData` object according to the managed method object.
Scenario I

- Compare the accessFlags in DexMethod with the access flag in the managed method object.

- Compare the codeoff in DexMethod with the code_item_off in the managed method object.

- If at least one is not equal, we modify the value in the DexMethod object according to the managed method object and write the relevant DexClassData into extra file.
Scenario II

◊ Check whether `code_item_off` exists in the scope of the dex file.

◊ If not, we collect the correct `code_item` and write it into `extra` file.
Reconstructing the Dex File

We now have four files: part1, classdef, data, extra.

We combine them as the sequence

(1) part1
(2) classdef
(3) data
(4) extra

Finally, we obtain a complete dex file.
Outline

- Background
- DexHunter
- Analysis of major products
- Related work
Products under Investigation

- 360  http://jiagu.360.cn/
- Ali http://jaq.alibaba.com/
- Baidu http://apkprotect.baidu.com/
- Bangcle http://www.bangcle.com/
- Tencent http://jiagu.qcloud.com/
- ijiami http://www.ijiami.cn/
Experiment Setup
<table>
<thead>
<tr>
<th></th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>360</td>
<td><code>/data/data/XXX/.jiagu/classes.dex</code></td>
</tr>
<tr>
<td>Ali</td>
<td><code>/data/data/XXX/files/libmobisecy1.zip</code></td>
</tr>
<tr>
<td>Baidu</td>
<td><code>/data/data/XXX/.1/classes.jar</code></td>
</tr>
<tr>
<td>Bangcle</td>
<td><code>/data/data/XXX/.cache/classes.jar</code></td>
</tr>
<tr>
<td>Tencent</td>
<td><code>/data/app/XXX-1.apk (/data/app/XXX-2.apk)</code></td>
</tr>
<tr>
<td>ijiami</td>
<td><code>/data/data/XXX/cache/.</code></td>
</tr>
</tbody>
</table>

XXX stands for its package name.
Anti-debugging

- All products detect debugger
- Anti-ptrace
- Anti-JWDP
- ....
- They cannot detect DexHunter.
♦ Version: 06-21-2015

♦ It encrypts the dex file and saves it in libjiagu.so/libjiagu_art.so.

♦ It releases the data into memory and decrypts it while running.
Version: 21-06-2015

It splits the original dex file into two parts

- One is the main body saved in libmobisecy.so
- The other one contains the `class_data_items` and the `code_items` of some `class_def_items`.

It releases both two parts into memory as plain text and corrects some offset values in the main body while running.

Some `annotation_offs` are set to incorrect values.
Version: 21-06-2015

It moves some class_data_items to other places outside the dex file.

It wipes the magic numbers, checksum and signature in the header after the dex file has been opened.
It fills in an empty method just before it is invoked and erases the content after the method is finished.

We instrument method invocation to dump these methods which is available only just before invoking.

- `DoInvoke` (ART)
- `dvmMterp_invokeMethod` (DVM)
Bangcle

- Version: 21-06-2015
- It prepares the odex file or oat file in advance.
- It encrypts the file and stores it in an external jar file.
- It decrypts the data while running
- It hooks several functions in libc.so, such as fwrite, mmap, ...
ijiami

❖ Version: 21-06-2015
❖ Similar to Bangcle
❖ The string changes every time the app runs.
❖ It releases the decrypted file, which is also encrypted as a jar file, with different file names each time while they are in the same directory.

- It can protect the methods selected by users.
- If a method is selected, it cannot be found in the relevant `class_data_item`.
- It releases the real `class_data_item` and adjusts the offset.
  - The `code_item` of the selected method is still in the data section.
- Some `annotation_offs` and `debug_info_offs` are set to 0xFFFFFFFF.
- It can only runs in DVM.
Outline

- Background
- DexHunter
- Analysis of Major Products
- Related work
Related work

thank you!